Building of systems of automatic C/C++ code logging

Andrey Karpov
OOO "Program Verification Systems"

May 2008

Abstract

Sometimes logging of an application's events is the only debugging method. The logging method's disadvantage is the large size of the code which you have to write manually to save the whole necessary information. The article touches upon the method allowing you to build a system of automatic logging of C/C++ code.

Introduction

Despite the current epoch C and C++ languages remain leaders in many programming spheres. And this will hardly change in nearest 10 years. A lot of nice and interesting languages have appeared but C and C++ are more preferable in practice. They are ones of the best existing languages - universal, high-performance and with wide support. If you want to learn why these languages are so attractive, I recommend you to get acquainted with the article "Tools and performance" [1] and an interesting collection of messages from comp.lang.c++.moderated forum "Why do you program in c++?" [2]. All this means that development of a toolkit for C/C++ languages is still topical.

In all the languages there are errors made by developers, and C/C++ are not an exclusion. Moreover, as well as many professional tools, they require more attention and accuracy. As the result one of the most important tasks while developing applications is code debugging, i.e. search and correction of errors.

Debugging methods can be divided into the following main groups:

- Interactive debugging tools;
- Runtime-diagnosis;
- Visual (graphical) debugging tools;
- Unit Testing;
- Functional tests;
- Logging;
- Crash-dumps debugging;
- Code review;
- Static code analysis.

Each of the mentioned methods has its advantages and disadvantages about which you may read in the article "Ways of debugging applications" [3]. But within the framework of our article we'll speak about logging and methods of its automation.

1. Why logging?

At first logging of applications' work may seem non-topical. Perhaps, it's just an atavism of those times when the results of a program's work would be immediately printed? No, this is a very effective and often very essential method allowing you to debug complex, parallel or specific applications.

Let's consider the spheres where logging is irreplaceable because of its convenience and efficiency:

1. Debugging of release-versions of an application. Sometimes a release-version behaves in a different way than a debug-version and it may be related to the errors of uninitialized memory etc. But it is often inconvenient to work with the release-version in debugger. And besides, although it happens quite seldom, there are compiler's errors which appear only in release-versions. Logging in these cases is a good substitute for compiler.
2. Debugging of security mechanisms. Development of applications with hardware security (for example, on the basis of Hasp keys) is often difficult because debugging is impossible here. Logging in this case seems to be the only way of searching for errors.
3. Logging is the only possible debugging method for an application, launched on the end user's computer. Accurate use of the log file will allow the developers to get the full information necessary for diagnosis of appearing problems.
4. Logging allows you to debug devices' drivers and programs for embedded systems.
5. Logging allows you to quicker detect errors after batch launch of functional or load tests.
6. One more interesting case of using logs is to view differences between two different versions (diff). The reader can think himself where in his projects it can be useful.
7. Logging enables remote debugging when interactive means are impossible or inaccessible. This is convenient for high-performance multi-user systems where the user puts his tasks into a queue and waits for them to be fulfilled. This approach is used nowadays in institutions and other organizations working with computing clusters.
8. Possibility to debug parallel applications. In such applications errors often occur when creating a lot of threads or when there are problems with synchronization. Errors in parallel programs are rather difficult to correct. A good method of detecting such errors is periodical logging of systems which relate to the error and examining of the log's data after program crash [1].

This list is too large to reject logging method. I hope that this article will help you to find other ways to use the described logging method with success.

2. Creation of a logging system

Let's begin with demands fulfilled by a modern logging system:

- The code providing logging of data in the debug-version shouldn't be present in the release-version of a program product. Firstly, it is related to speeding up performance and decreasing the size of the program product. Secondly, it makes impossible to use the logging information for cracking the application or other illegal actions. Pay attention that it is the final version of the program that is meant as the log may be created by both debug- and release-versions.
- Logging system's interfaces should be compact in order not to overload the main program code.
- Saving of data should be carried as quickly as possible in order to bring the minimum change into temporary characteristics of parallel algorithms.
- The received log should be understandable and easy to analyze. There should be a possibility to divide information received from different threads and to vary the number of its details.

- Besides logging of the application's events themselves, it is useful to collect data about the computer.
- It is desirable that the system save the unit's name, the file's name and the string's number where data record occurred. Sometimes it is useful to save the time when the event took place.

A logging system meeting these demands allows you to fulfill various tasks from developing security mechanism to searching for errors in parallel algorithms [4].

Although this article is devoted to a system of data logging, it won't touch upon some particular complete version of such a system. The universal version is impossible as it much depends upon the development environment, the project's peculiarities, the developer's preferences etc. By the way, consider some articles concerning these questions: "Logging in C++" [5] and "Ways of debugging applications: Logging" [6].

Now let's speak about some technical solutions which will help you to create a convenient and efficient logging system when you need it.

The simplest way to carry out logging is to use a function similar to printf as in the following example:

```
  int x = 5, y = 10;
  ...
  printf("Coordinate = (%d, %d)\n", x, y);
```

The natural disadvantage here is that information will be shown both in the debug-mode and in the release-version. That's why we should modify the code in the following way:

```
#ifdef DEBUG_MODE
  #define WriteLog printf
#else
  #define WriteLog(a)
#endif

  WriteLog("Coordinate = (%d, %d)\n", x, y);
```

This is better. And pay attention that to implement WriteLog function we use our own macro DEBUG_MODE instead of the standard _DEBUG. This allows you to include logging information into release-versions what is important when debugging at large data size.

Unfortunately, now when compiling a non-debug version, for example in Visual C++ environment, a warning message appears: "warning C4002: too many actual parameters for macro 'WriteLog'". You may disable this warning but it will be a bad style. You can rewrite the code as follows:

```
#ifdef DEBUG_MODE
  #define WriteLog(a) printf a
#else
  #define WriteLog(a)
#endif

  WriteLog(("Coordinate = (%d, %d)\n", x, y));
```

This code is not smart as you have to use double bracket pairs and you can lose them. That's why let's improve it a bit:

```
#ifdef DEBUG_MODE
  #define WriteLog printf
```

```
#else
  inline int StubElepsisFunctionForLog(...) { return 0; }

  static class StubClassForLog {
  public:
    inline void operator =(size_t) {}
  private:
    inline StubClassForLog &operator =(const StubClassForLog &)
      { return *this; }
  } StubForLogObject;

  #define WriteLog \
    StubForLogObject = sizeof StubElepsisFunctionForLog
#endif

  WriteLog("Coordinate = (%d, %d)\n", x, y);
```

This code looks complicated but it allows you to write single brackets. When DEBUG_MODE is disabled this code turns to nothing and you can safely use it in critical code sections.

The next improvement is to add to the logging function such parameters as the number of details and the type of printed information. The number of details can be defined as a parameter, for example:

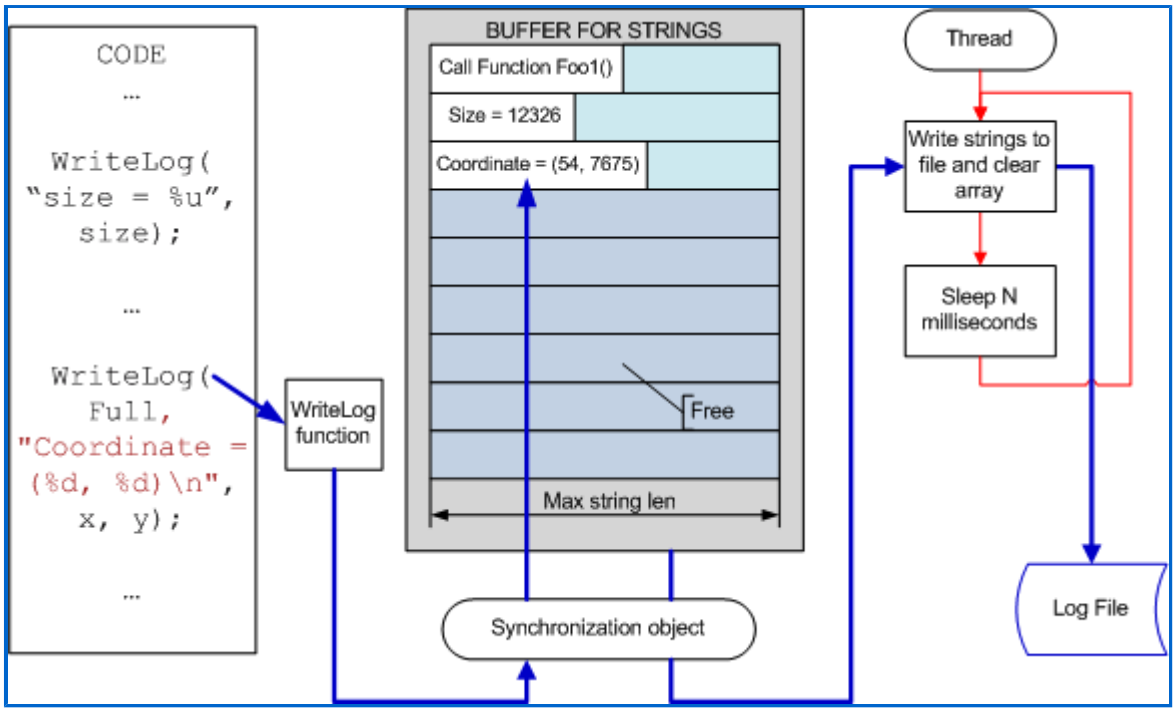
```
enum E_LogVerbose {
  Main,
  Full
};

#ifdef DEBUG_MODE
  void WriteLog(E_LogVerbose,
                const char *strFormat, ...)
  {
    ...
  }
#else
  ...
#endif

WriteLog (Full, "Coordinate = (%d, %d)\n", x, y);
```

This method is convenient because the decision whether to filter unimportant messages or not can be made after the program's shutdown by using a special utility. This method's disadvantage is that the full information is always printed, both important and unimportant what may decrease the performance. That's why you may create several functions of WriteLogMain- and WriteLogFull-type and similar to them, implementation of which will depend upon the program build mode.

We mentioned that record of logging information must influence the speed of the algorithm's work as little as possible. You may get it by creating a system of collecting messages which are recorded in a parallel thread. Now it has even more advantages because of wide spread of multi-core (multi-processor) systems. The scheme of this mechanism is shown on picture 1.

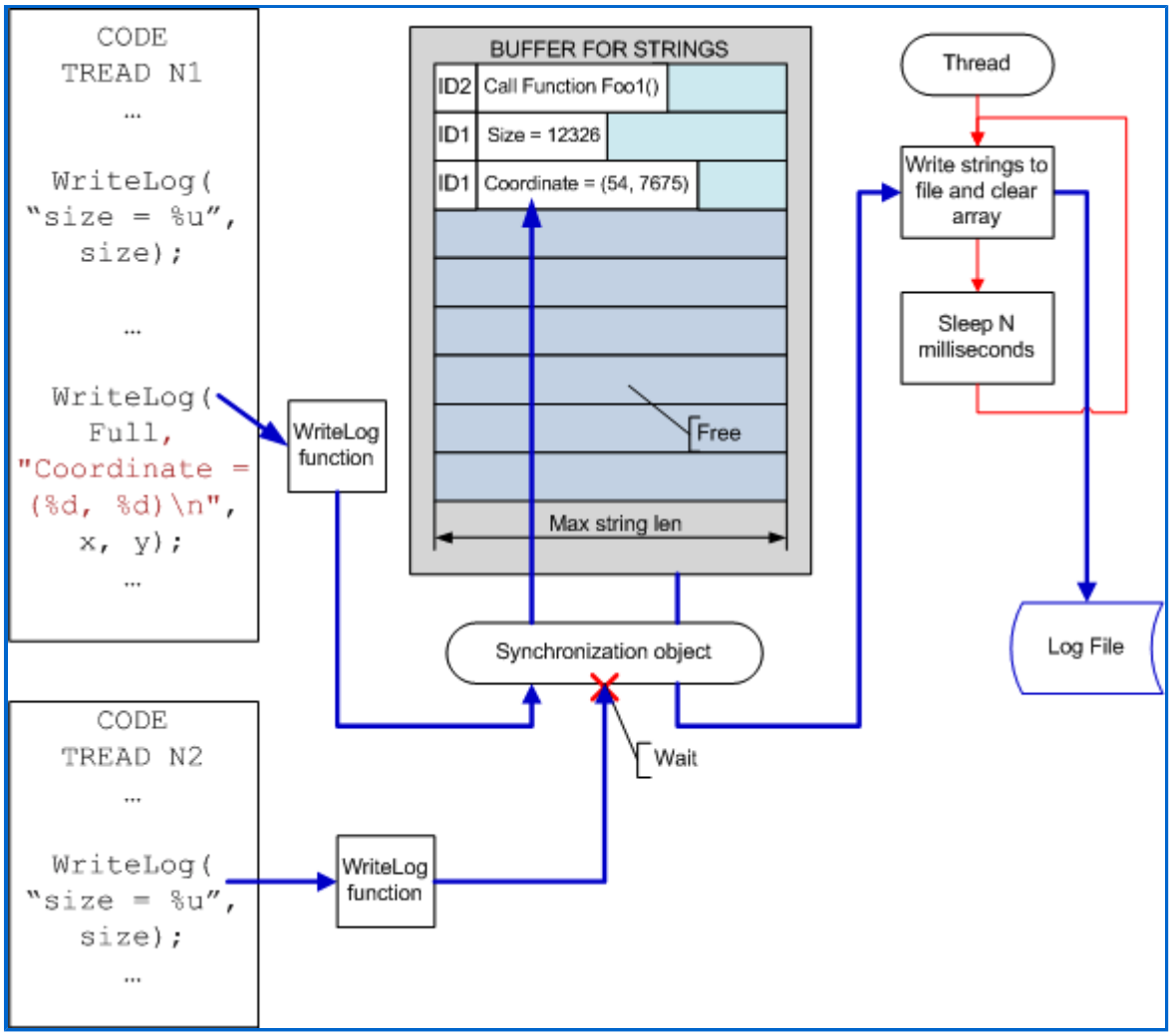


Picture 1.Logging system with delayed data record.

As you may see on the picture, the next data portion is recorded into an intermediate array with strings of fixed length. The fixed length of the array and its strings allows you to avoid expensive operation of memory allocation. It doesn't decrease this system's possibilities at all. You can just select the strings' length and the array's size with some reserve. For example, 5000 strings of 4000-symbol length will be enough for debugging nearly any system. I think you agree that 20-MB memory size necessary for this is not critical for modern systems. But if an overflow of the array still occurs, you can easily create a mechanism of anticipatory information record into the file.

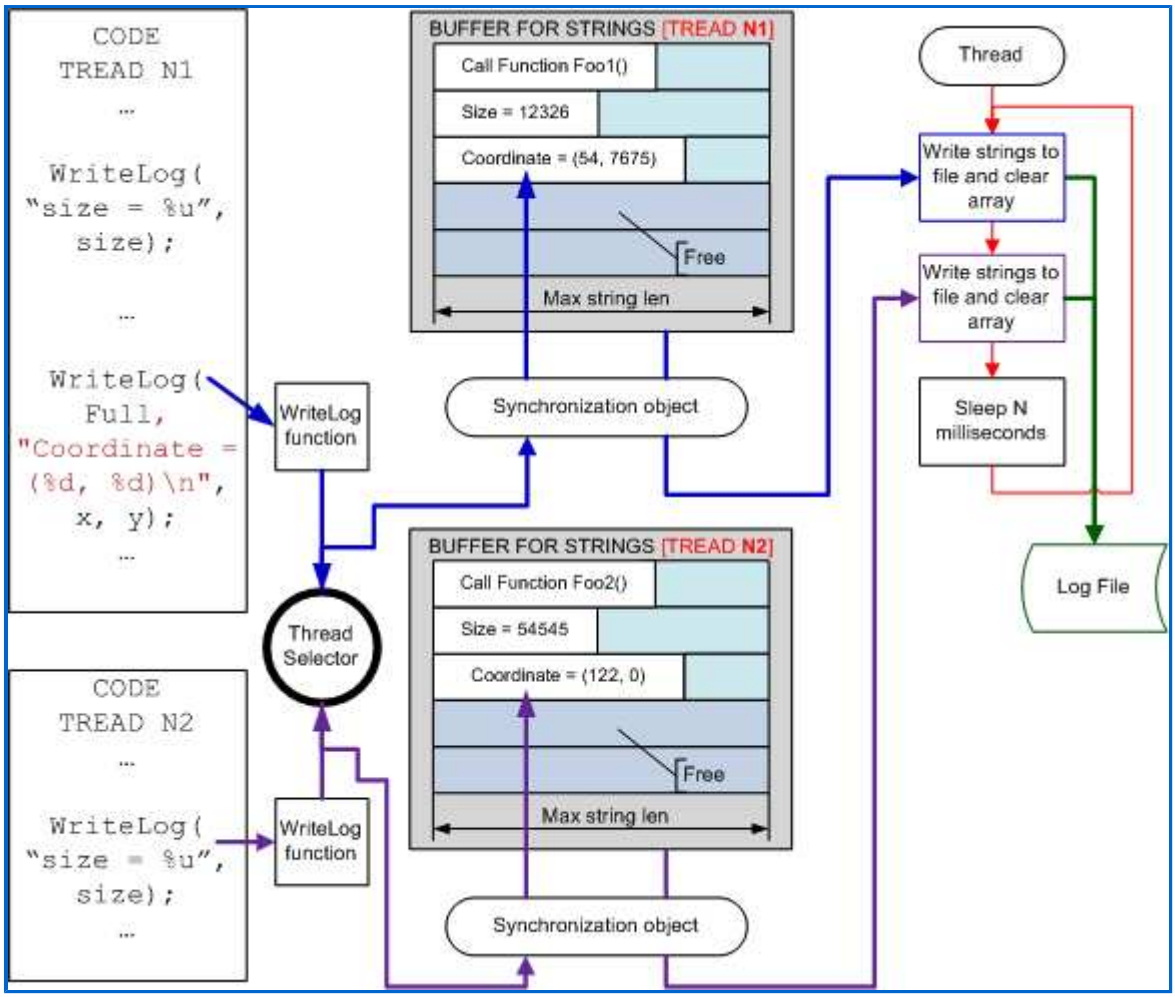
This mechanism provides nearly instant execution of WriteLog function. If there are unloaded processor cores in the system, the file record will be transparent for main threads of the program being logged.

The advantage of the described system is that it can work practically without changes when debugging a parallel program, when several threads are written into the log simultaneously. You should only add saving of a thread-indicator so that you can later see from which threads messages were received (see Picture 2).



Picture 2. Logging system when debugging multithread applications.

This scheme can cause change of temporal characteristics as several threads being logged have to wait for each other to carry out information record. If it is critical you may create separate message storages for each of the threads as is shown on Picture 3. In this case you should also record the time of the events so that you could later merge two logs into one.



Picture 3. Improved logging system when debugging multithread applications.

The last improvement I would like to offer is providing demonstration of message nesting level when calling functions or in the beginning of a logical block. You can easily implement it by using a special class which writes the block's beginning identifier into the log in the constructer and the block's end identifier in the destructor. By writing a small utility you may transform the log relying on the information about identifiers. Let's try to show it by example.

Program code:

```
class NewLevel {
public:
  NewLevel() { WriteLog("__BEGIN_LEVEL__\n"); }
  ~NewLevel() { WriteLog("__END_LEVEL__\n"); }
};

#define NEW_LEVEL NewLevel tempLevelObject;


void MyFoo() {
  WriteLog("Begin MyFoo()\n");
  NEW_LEVEL;
  int x = 5, y = 10;
  printf("Coordinate = (%d, %d)\n", x, y);
  WriteLog("Begin Loop:\n");
  for (unsigned i = 0; i != 3; ++i)
  {
    NEW_LEVEL;
    WriteLog("i=%u\n", i);
```

```
  }
}
```

The log's content:

```
Begin MyFoo()
__BEGIN_LEVEL__
Coordinate = (5, 10)
Begin Loop:
__BEGIN_LEVEL__
i=0
__END_LEVEL__
__BEGIN_LEVEL__
i=1
__END_LEVEL__
__BEGIN_LEVEL__
i=2
__END_LEVEL__
Coordinate = (5, 10)
__END_LEVEL__
```

The log after transformation:

```
Begin MyFoo()
    Coordinate = (5, 10)
    Begin Loop:
        i=0
        i=1
        i=2
    Coordinate = (5, 10)
```

3. Automation of logging system

We have considered principles on which a logging system can be implemented. All the demands described in the first part can be implemented in such a system. But a serious disadvantage of this system is the necessity to write a great amount of code for recording all the necessary data. There are other disadvantages too:

1. Impossibility to implement smart logging functions for C language as it lacks classes and templates. As the result the code for logging is different for C and C++ files.
2. Necessity to keep in mind that you should write NEW_LEVEL in every block or function if you wish your code look smart.
3. Impossibility to automatically save names of all called functions. Necessity to manually write the input function arguments.
4. Overload of source texts with additional constructions as, for example, NEW_LEVEL.
5. Necessity to make an effort to control that all the logging constructions should be excluded from the final version of the program.
6. Necessity to write functions for initializing a logging system, to write all the necessary "#include" and perform other auxiliary actions.

All these and other inconveniences can be avoided if you build a system of automatic logging of testing of applications.

This system can be implemented on the basis of metaprogramming method, introducing new constructions of data record into C/C++ language.

Metaprogramming is creation of programs which in their turn create other programs [7]. There are two main trends in metaprogramming: code generation and self-modified code. We are interested in the first one. In this case the program code with embedded logging mechanisms is not written manually but created automatically by a generator-program on the basis of another, simpler program. This allows you to get a program at less time and effort costs than in the case when a programmer implements the whole code himself.

There are languages in which metaprogramming is their constituent part. Such an example is Nemerle language [8]. But it is more difficult for C/C++ and metaprogramming in them is implemented in two following ways:

1. Templates in C++ and preprocessor in C. Unfortunately, as was shown before, it's not enough.
2. Outer language means. The generator's language is composed in the way to automatically or with a programmer's minimum effort implement paradigm's rules or necessary special functions. In fact, a more high-level programming language is created. It is this approach that can be used for creation of a system of automated program logging.

By introducing new keywords into C/C++ you can get a flexible and convenient programming system with very powerful logging possibilities. Use of metalanguage provides great possibilities of choosing the form in which the necessary information will be recorded. You may log the data by introducing the common function format:

```
EnableLogFunctionCallForFhisFile(true);
...
ObjectsArray &objects = getObjects();
WriteLog("objects.size = %u", objects.size());
for (size_t i = 0; i != objects.size(); ++i) {
    WriteLog("object type = %i", int(objects[i]->getType()));
    if (objects[i]->getType() != TYPE_1)
          ...
```

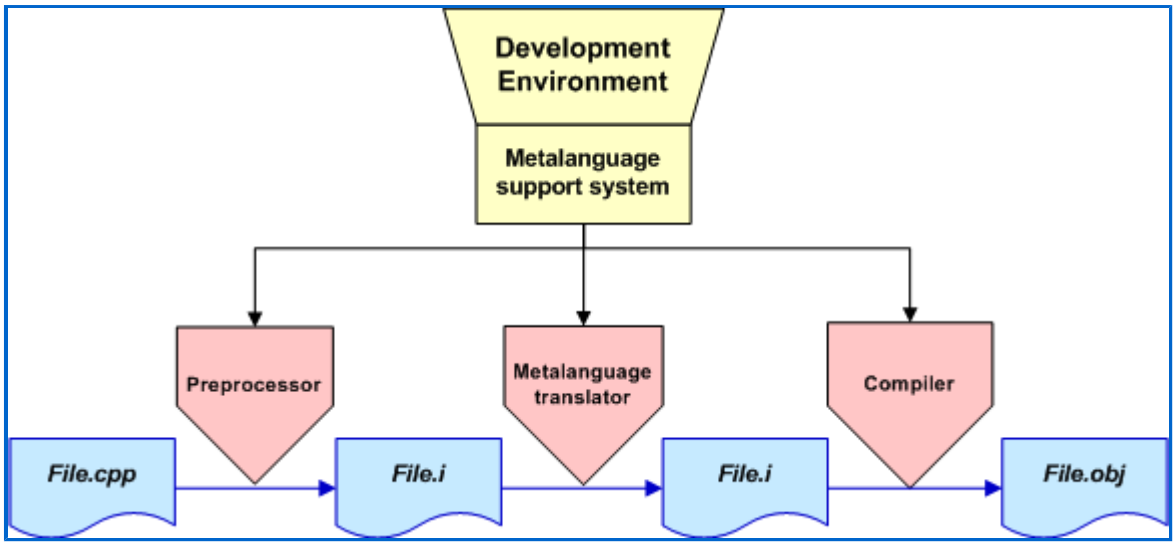
Since the code generator can recognize the data types being used, you may simplify logging of variables and shorten the record. For example:

```
ObjectsArray &objects = getObjects();
MyLog "objects.size = ", objects.size()
for (size_t i = 0; i != objects.size(); ++i) {
    MyLog "object type = ", int(objects[i]->getType())
    if (objects[i]->getType() != TYPE_1)
          ...
```

You may go further as everything here depends upon the author's fantasy:

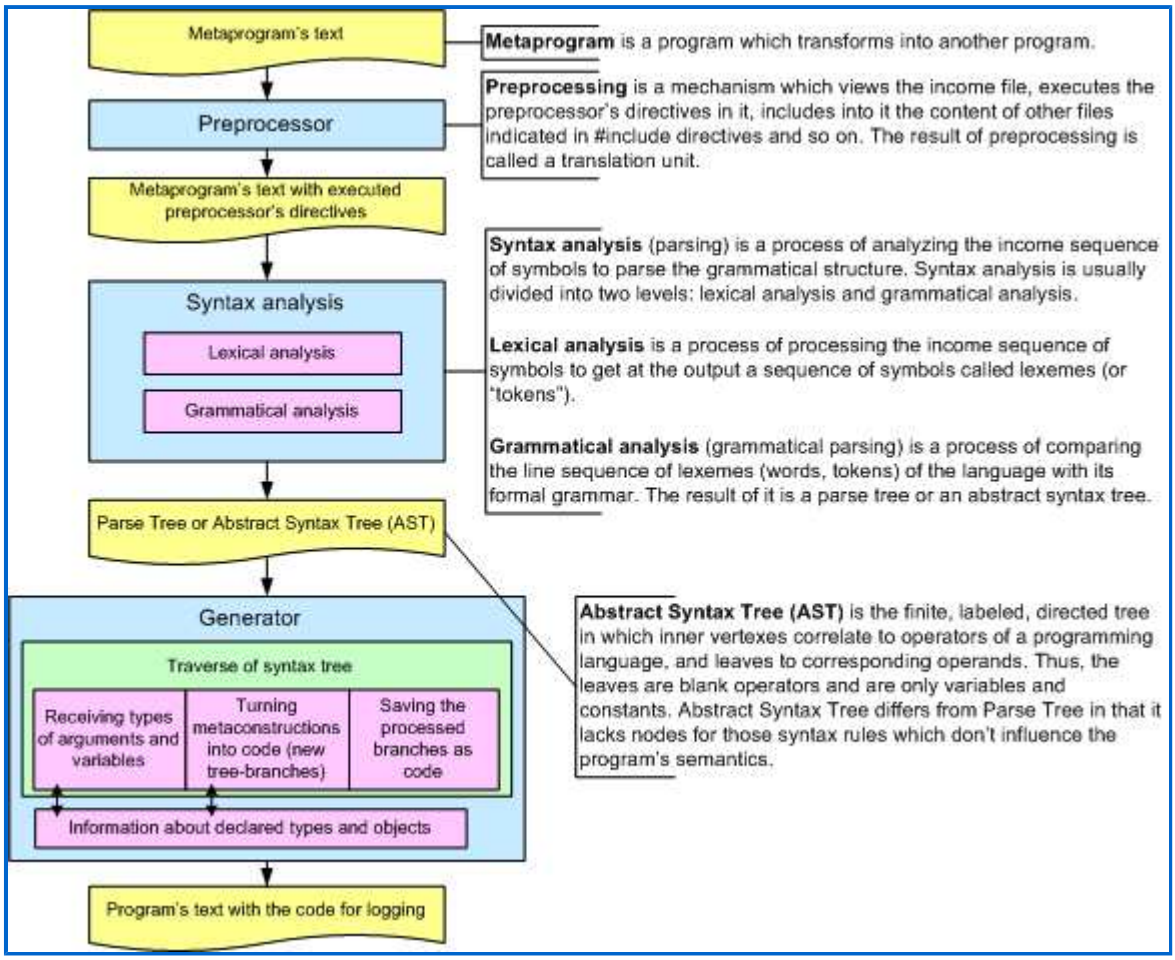
```
thisfile.log.FunctionCall = ON
...
ObjectsArray &objects = getObjects();
for (size_t i = 0;
    i != <log>"objects.size = "objects.size()<log>;
    ++i) {
    LOG: "object type = " {as int} objects[i]->getType();
    if (objects[i]->getType() != TYPE_1)
          ...
```

The new metalanguage can be implemented as an intermediate constituent between a preprocessor and a compiler. It depends upon the development environment how this intermediate component for translating a metaprogram should look. In general, the functioning scheme looks as on Picture 4.



Picture 4. Role of metalanguage translator in the compilation process.

Now let's consider the translator itself. Its functional scheme is shown on Picture 5.



Picture 5. Functional scheme of language translator for generating code with logging support.

Let's dwell upon the code generator. It is an algorithm of traversing the tree and carrying out three operations over it. The first is collecting types of all the functions and objects and estimating their visibility scope. It will allow you to automatically generate code for correct record of arguments of functions and other objects. Secondly, metalanguage constructions are opened into new tree branches. And thirdly, processed code branches are recorded back into the file.

It is not so simple to implement such a generator as well as a syntactic analyzer. But there are corresponding libraries which will be described further.

Let's see how generation of program code solves the above mentioned disadvantages of a system built only on the basis of macros and templates.:

1. Logging constructions in C may look as smart as in C++. The generator performs the duty of turning reserved words into code for writing messages. It is transparent for the user that this code will look in different ways in C and C++ files.
2. The generator can automatically insert code for marking the beginning and the end of functions or blocks. There is no need to use NEW_LEVEL macro.
3. You may automatically save the names of all or some of the functions being called. You may also automatically save values of input parameters for basic data types.
4. The text is no more overloaded with auxiliary constructions.
5. It is guaranteed that all the functions, sources and objects won't be present in the final version of the program product. The generator can just skip all the special elements related to logging in the program text.
6. There is no need to write initializing functions, to insert "#include" and perform other auxiliary actions as they can be implemented at the step of code translation.

But this scheme has a problem of debugging of such modified programs with the help of a debugger. On the one hand, if there is a logging system, the debugger is not an important component of application development. But on the other hand, it is often a very useful tool and you don't want to refuse it.

The trouble is that after code translation (opening of operators for logging) there is a problem of navigation on strings' numbers. It can be solved with the help of specific program means which are individual for each development environment. But there is a simpler way. You may use the approach similar to one in OpenMP, i.e. use "#pragma". In this case the logging code will look like this:

```
ObjectsArray &objects = getObjects();
#pragma Log("objects.size = ", objects.size())
for (size_t i = 0; i != objects.size(); ++i) {
    #pragma Log("object type = ", objects [i]->getType())
    if (objects[i]->getType() != TYPE_1)
          ...
```

It's not so smart because of the word "#pragma" but this program text has great advantages. This code can be safely compiled on another system where a system of automatic logging is not used. This code can be ported on another system or be given to third-party developers. And of course there are no obstacles in working with this code in the debugger.

4. Toolkit

Unfortunately, I don't know if there are metalanguage-based tools of automated logging similar in functionality to those described in the article. If further research shows the absence of such developments, perhaps I will take part in a new project which is being designed now. One may say this article is a kind of research whether a universal logging system for C/C++ languages is topical.

And for the present I offer the readers who got interested to create a logging system by themselves. Whether it is just a set of templates and macros or a full system of code generation depends upon the need for such a system and the amount of resources necessary to create it.

If you will stop at creation of a metalanguage we'll advise you on what basis it can be created.

There are two most suitable free libraries allowing you to represent a program as a tree, translate it and save it as a program again.

One of them is OpenC++ (OpenCxx) [9]. It is an open free library which is a "source-to-source" translator. The library supports metaprogramming and allows you to create on its basis extensions of C++ language. On the basis of this library such solutions were created as an execution environment OpenTS for T++ programming language (product by Institution of program systems RAS) and Synopsis tool for preparing documentation on the source code.

The second library is VivaCore which is a development of OpenC++ library oriented on Microsoft Visual Studio 2005/2008 development environment [10]. This library is also open and free. Its main differences are implemented support of C language and of some modern elements of C++ language. Static analyzers Viva64 and VivaMP are examples how such a library can be used.

If you would like to use these libraries to create your own extensions you will need a preprocessor that must be launched before they start working. Most likely, preprocessing means embedded into the compiler will be enough. Otherwise you may try to use The Wave C++ preprocessor library.

Conclusion

The article is of a theoretical character but I hope that developers will find here a lot of interesting and useful ideas related to the sphere of logging of their programs' code. Good luck!

Sources

1. Scott Bilas. Tools & Productivity. Casual Connect Magazine (Winter 2007) - http://www.viva64.com/go.php?url=108.
2. Over at comp.lang.c++.moderated, there is a thread created by the c++ guru Scott Meyers on the subject: "Why do you program in c++?" http://www.viva64.com/go.php?url=35
3. Dmitriy Dolgov. Ways of debugging applications. http://www.viva64.com/go.php?url=36
4. Andrey Karpov, Evgeniy Ryzhkov. Development of resource-intensive applications in Visual C++ http://www.viva64.com/art-1-2-2014169752.html
5. Petru Marginean. Logging In C++. http://www.viva64.com/go.php?url=37
6. Dmitriy Dolgov. Methods of debugging applications: Logging. http://www.viva64.com/go.php?url=38
7. Jonathan Bartlett. The art of metaprogramming, Part 1: Introduction to metaprogramming. http://www.viva64.com/go.php?url=109
8. Nemerle. http://nemerle.org/
9. Grzegorz Jakack. OpenC++ - A C++ Metacompiler and Introspection Library. http://www.viva64.com/go.php?url=41
10. Andrey Karpov, Evgeniy Ryzhkov. The essence of the code analysis library VivaCore. http://www.viva64.com/art-2-2-449187005.html

About the author

Andrey Karpov, Candidate of Physico-mathematical Sciences, science consultant of OOO "Program Verification Systems" specializing in questions of increasing quality of program systems. Solves theoretical and practical questions of static code analysis. Takes part in creation of VivaCore library and code analyzers Viva64, VivaMP.

Page on LinkedIn site: http://www.linkedin.com/pub/4/585/6a3

E-Mail: karpov@viva64.com
